COMP 6231: Distributed System Design

Web Services

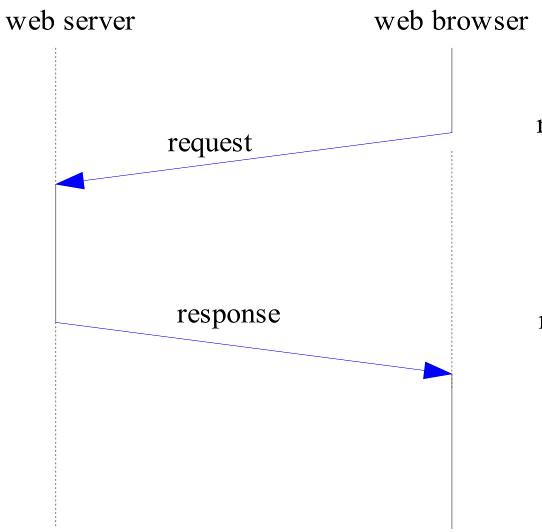
Based on Chapter 9 of the textbook and the slides from
Prof. Kenneth P. Birman, Cornell University,
Prof. M.L. Liu, California Polytechnic State University

The HyperText Transfer Protocol (HTTP)

- HTTP is a connection-oriented, stateless, request-response protocol.
- An HTTP server, or web server, runs on TCP port 80 by default.
- HTTP clients, colloquially called web browsers, are processes which implements HTTP to interact with a web server to retrieve documents phrased in HTML, whose contents are displayed according to the documents' markups.

The HyperText Transfer Protocol (HTTP)

- In HTTP/1.0, each connection allows only one round of request-response.
- A client obtains a connection, issues a request
- The server processes the request, issues a response, and closes the connection thereafter.



request is a message in 3 parts:
- <command> <document adddress> <HTTP version>
- an optional header
- optional data for CGI data using post method

response is a message consisting of 3 parts:
- a status line of the format <protocol><status code><description>
- header information, which may span several lines;
- the document itself.

The HTTP request

- A client request is sent to the server after the client has established a connection to the server.
- A request line is of the following form:

```
<HTTP method><space><RequestURI><space>
         <protocol specification>\r\n
```

where

  - `<HTTP method>` is the name of a method defined,
  - `<Request-URI>` is the URI of a web document, or, more generally, a web object,
  - `<protocol specification>` is a specification of the protocol observed by the client, and
  - `<space>` is a space character.

- An example client request is as follows:

```
GET /index.html HTTP/1.0
```

HTTP Methods in a client request

- The HTTP method in a client request is a reserved word (in uppercase) which specifies an operation of the server that the client desires.
- Some of the key client request methods are listed below:
  - GET: for retrieving the contents of web object referenced by the specified URI
  - HEAD: for retrieving a header from the server only, not the object itself.
  - POST: used to send data to a process on the server host.
  - PUT: used to request the server to store the contents enclosed with the request to the server machine in the file location specified by the URI.

The Request Header

- Some of the keywords and values that may appear in a request header are:
  - Accept: content types acceptable by the client
  - User-Agent: specifies the type of browser
  - Connection: “Keep-Alive” can be specified so that the server does not immediately close a connection after sending a response.
  - Host: host name of the server
- An example request header is as follows:

```
Accept: */*
Connection: Keep-Alive
Host: www.someU.edu
User-Agent: Generic
```

HTTP Response Header

- **Response header lines** – these header lines return information about the response, the server, and further access to the resource requested, as follows:

```
Age: seconds
Location: URI
Retry-After: date|seconds
Server: string
WWW-Authenticate: scheme realm
```

- **Entity header lines** – these header lines contain information about the contents of the object requested by the client, as follows:

```
Content-Encoding
Content-Length
Content-Type: type/subtype (see MIME)
Expires: date
Last-Modified: date
```

HTTP Response Header

- An Example response header is as follows:

```
Date: Mon, 30 Oct 2000 18:52:08 GMT
Server: Apache/1.3.9 (Unix) ApacheJServ/1.0
Last-modified: Mon, 17 June 2001 16:45:13 GMT
Content-Length: 1255
Connection: close
Content-Type: text/html
```

- The Content-Type specifies the type of the data, using the contents type designation of the MIME protocol.
- The Content-Encoding specifies the encoding scheme (such as ***uuencode*** or ***base64***) of the data, usually for the purpose of data compression.
- The expiration date gives the date/time (specified in a format defined with HTTP)after which the web object should be considered stale
- The Last-Modifed date specifies the date that the object was last modified.

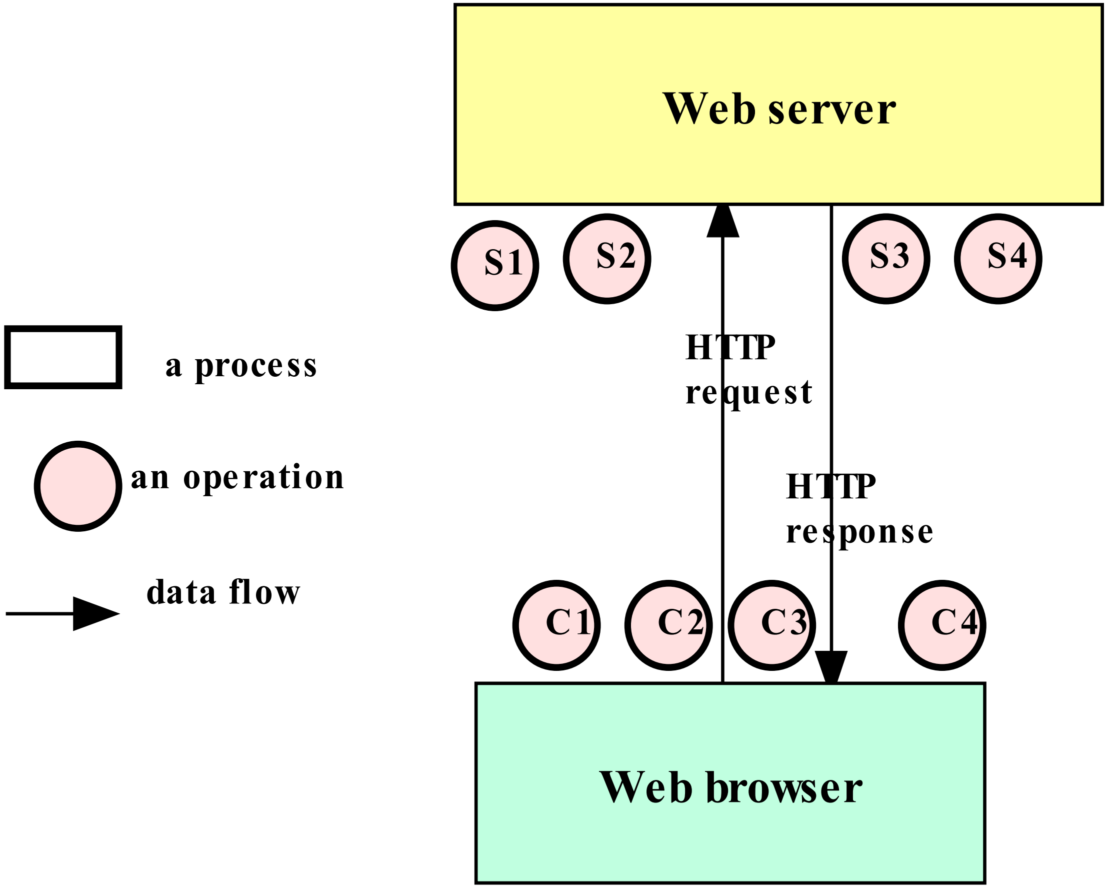
HTTP Response Body

The body of the response follows the header and a blank line, and contains the contents of the web object requested.

```
HTTP/1.1 200 OK
Date: Sat, 15 Sep 2001 06:55:30 GMT
Server: Apache/1.3.9 (Unix) ApacheJServ/1.0
Last-Modified: Mon, 30 Apr 2001 23:02:36 GMT
ETag: "5b381-ec-3aedef0c"
Accept-Ranges: bytes
Content-Length: 236
Connection: close
Content-Type: text/html

<html>
<head>
<title>My web page </title>
</head>
<body>
Hello world!
</BODY></HTML>
```

Interprocess Communication in basic HTTP



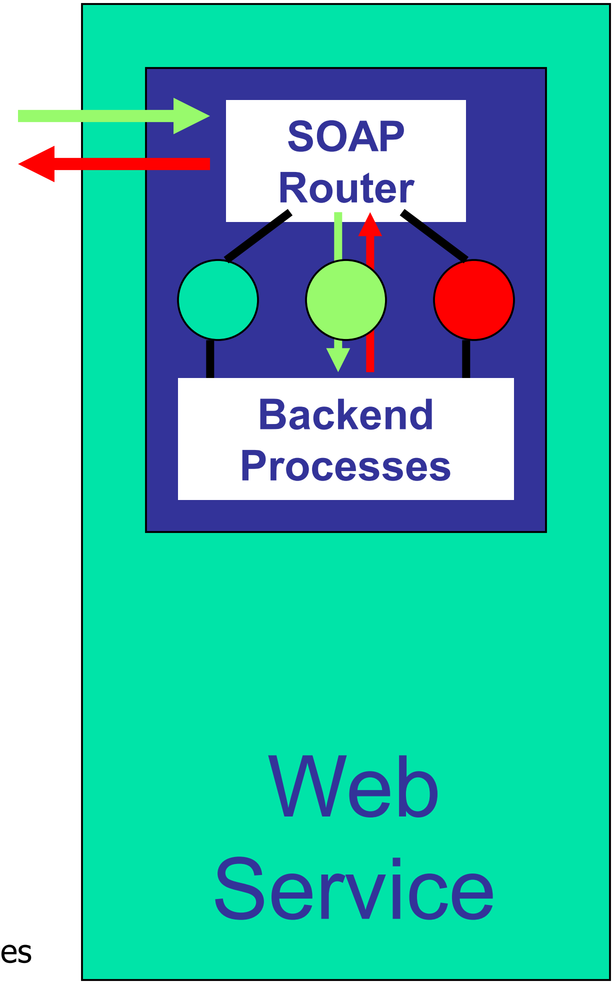
operations:
S1: accept connection
S2: receive (request)
S3: send (response)
S3: disconnect
C1: make connection
C2: send (request)
C3: receive (response)
C4: disconnect

What are Web Services?

- Today, we normally use Web browsers to talk to Web sites
  - Browser names document via URL (lots of fun and games can happen here)
  - Request and reply encoded in HTML, using HTTP to issue request to the site
- Web Services generalize this model so that computers can talk to computers

What are Web Services?

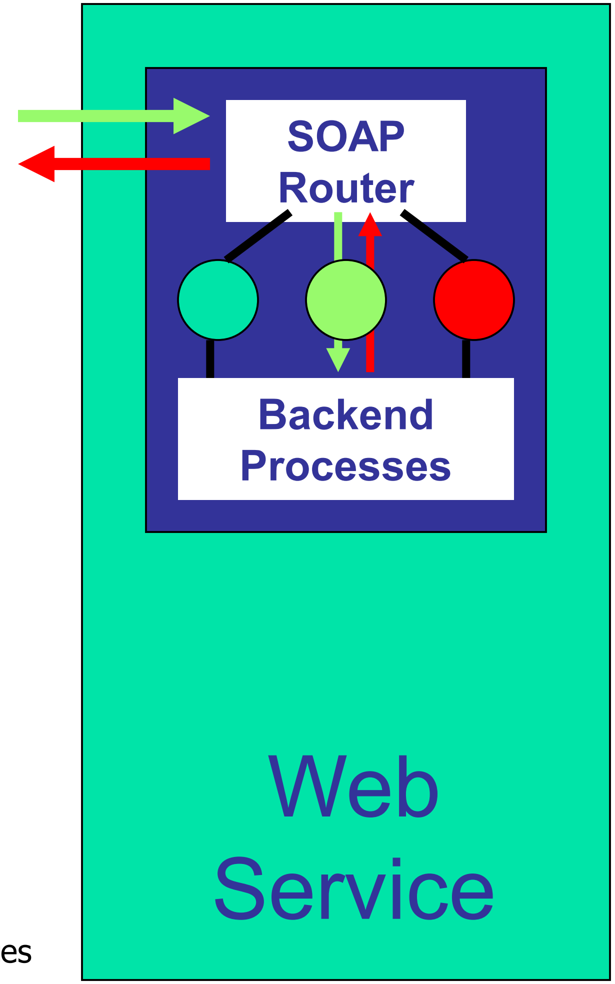
- "Web Services are software components described via WSDL which are capable of being accessed via standard network protocols such as SOAP over HTTP."



What are Web Services?

- “Web Services are software components described via WSDL which are capable of being accessed via standard network protocols such as SOAP over HTTP.”

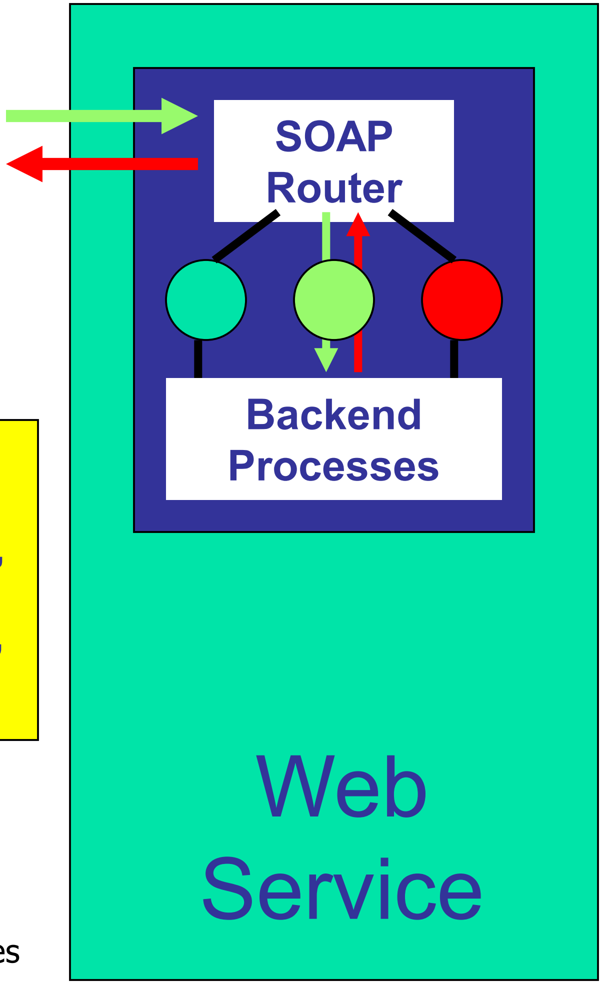
Today, SOAP is the primary standard. SOAP provides rules for encoding the request and its arguments.



What are Web Services?

- "Web Services are software components described via WSDL which are capable of being accessed via standard network protocols such as SOAP over HTTP."

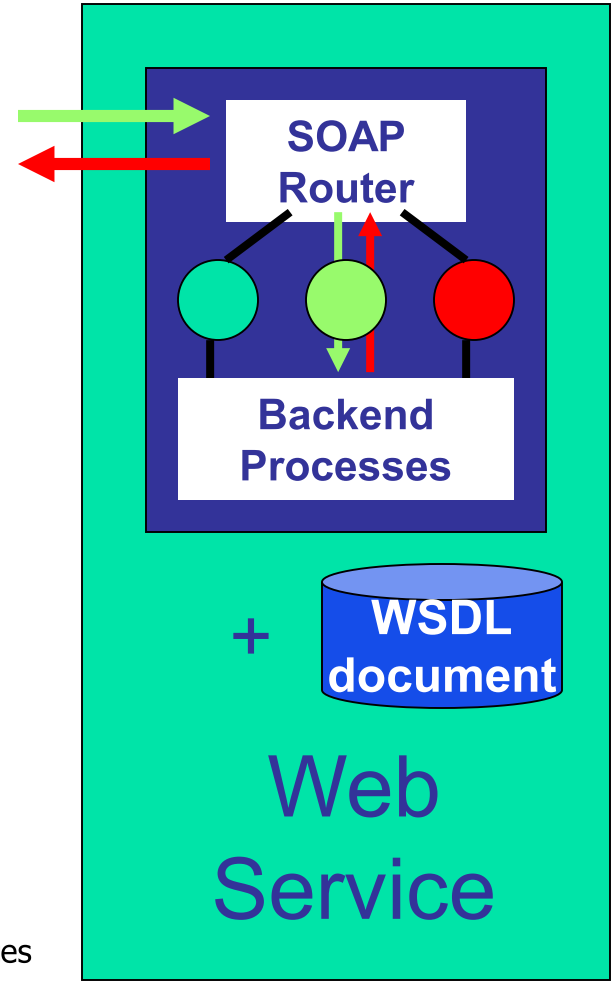
Similarly, the architecture doesn't assume that all access will employ HTTP over TCP. In fact, .NET uses Web Services "internally" even on a single machine. But in that case, communication is over COM



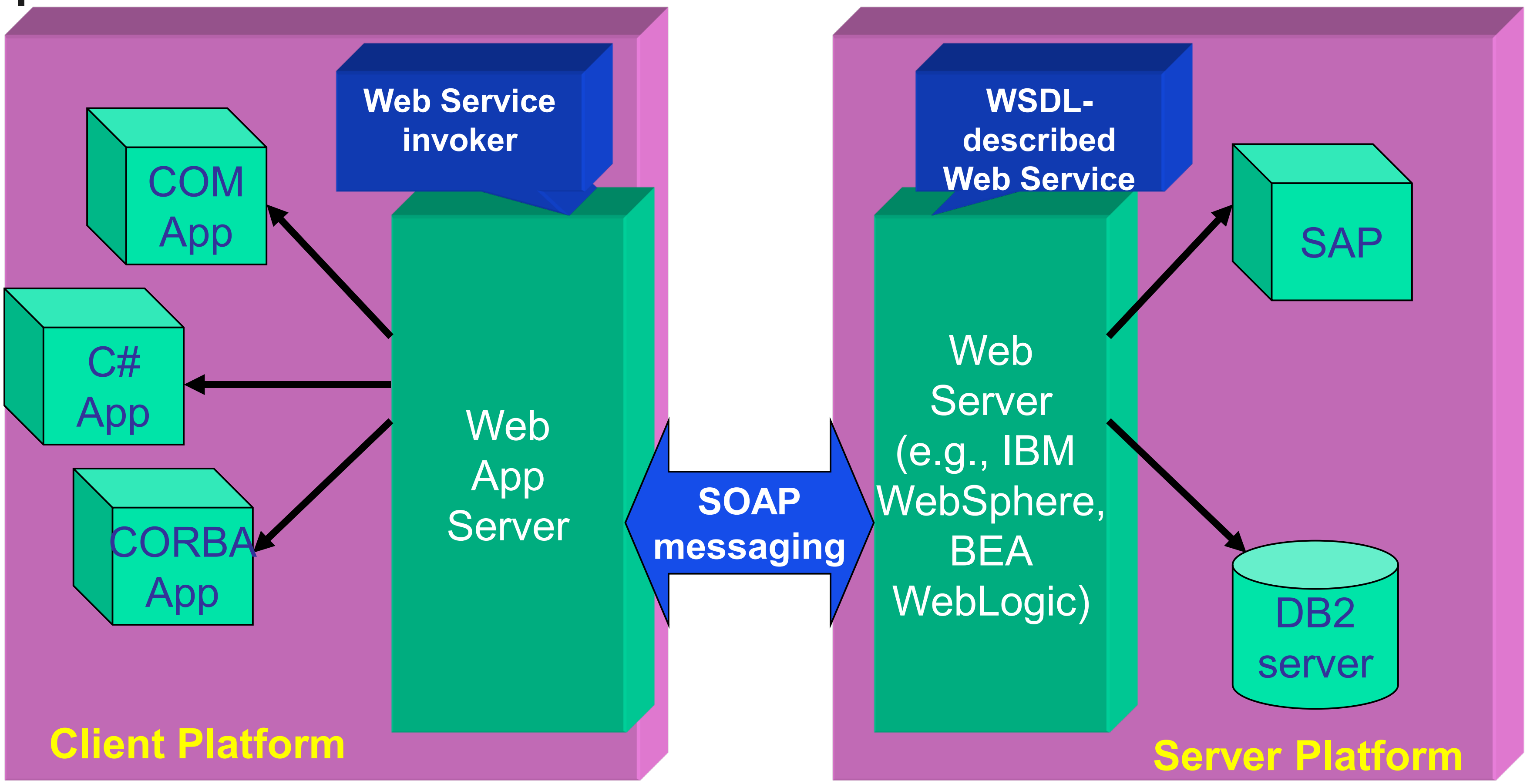
What are Web Services?

- "Web Services are software components **described via WSDL** which are capable of being accessed via standard network protocols such as SOAP over HTTP."

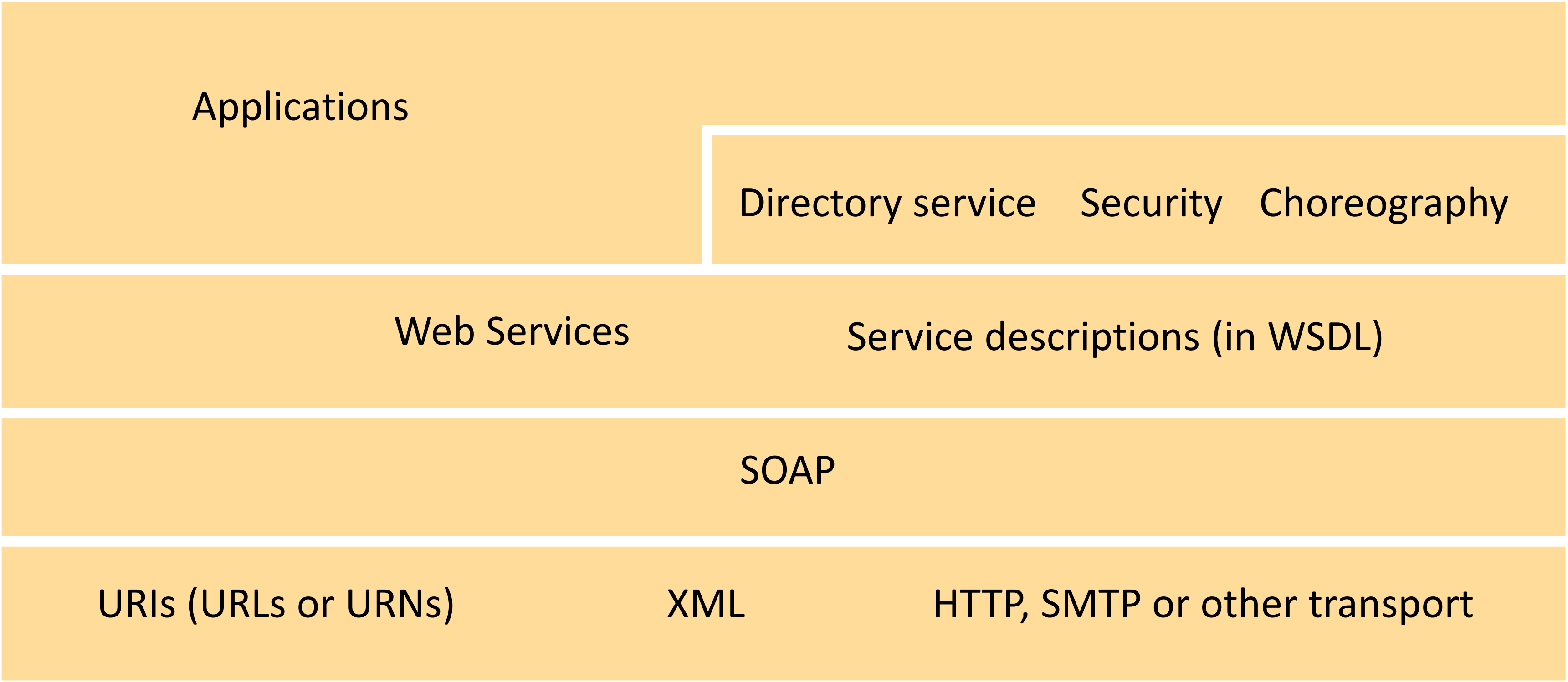
WSDL documents are used to drive object assembly, code generation, and other development tools.



Web Services are often Front Ends



Web services infrastructure and components



Applications

Directory service Security Choreography

Web Services Service descriptions (in WSDL)

SOAP

URIs (URLs or URNs) XML HTTP, SMTP or other transport

Advantages of web services?*

- Web services provide interoperability between various software applications running on various platforms.
  - "vendor, platform, and language agnostic"
- Web services leverage open standards and protocols. Protocols and data formats are text based where possible
  - Easy for developers to understand what is going on.
- By piggybacking on HTTP, web services can work through many common firewall security measures without requiring changes to their filtering rules.

*: **From Wikipedia**

How Web Services work

- First the client *discovers* the service.
- Typically, client then *binds* to the server.
  - By setting up TCP connection to the discovered address .
  - But binding not always needed.
- Next build the SOAP request: (*Marshaling*)
  - Fill in what service is needed, and the arguments. Send it to server side.
  - XML is the standard for encoding the data (but is very verbose and results in HUGE overheads)

How it works…

- SOAP router routes the request to the appropriate server (assuming more than one available server)
  - Can do load balancing here.
- Server unpacks the request (*Unmarshaling*), handles it, computes result.
- Result sent back in the reverse direction: from the server to the SOAP router back to the client.

Discovery

- This is the problem of finding the “right” service
  - In our example, we saw one way to do it – with a URL
  - Web Services community favors what they call a URN: Uniform Resource Name
- But the more general approach is to use an intermediary: a discovery service

Example of a repository

Name	Type	Publisher	Toolkit	Language	OS
Web Services Performance and Load Tester	Application	LisaWu		N/A	Cross-Platform
Temperature Service Client	Application	vinuk	Glue	Java	Cross-Platform
Weather Buddy	Application	rdmgh724890	MS .NET	C#	Windows
DreamFactory Client	Application	billappleton	DreamFactory	Javascript	Cross-Platform
Temperature Perl Client	Example Source	gfinke13		Perl	Cross-Platform
Apache SOAP sample source	Example Source	xmethods.net	Apache SOAP	Java	Cross-Platform
ASS 4	Example Source	TVG	SOAPLite	N/A	Cross-Platform
PocketSOAP demo	Example Source	simonfell	PocketSOAP	C++	Windows
easysoap temperature	Example Source	a00	EasySoap++	C++	Windows
Weather Service Client with MS- Visual Basic	Example Source	oglimmer	MS SOAP	Visual Basic	Windows
TemperatureClient	Example Source	jgalyan	MS .NET	C#	Windows

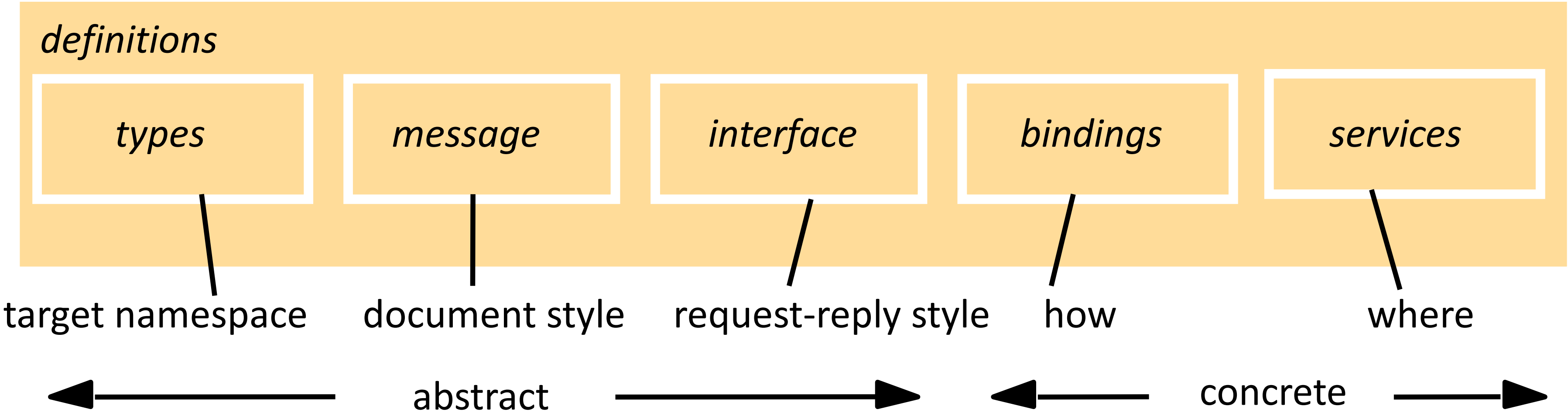
Repository summary

- A database listing servers
- Each is described using the UDDI language, which is defined over XML
  - Hence can be searched with XML queries
- An extensible standard
  - Defines some required information about interfaces available and argument types, etc
  - But services can provide extra information too.

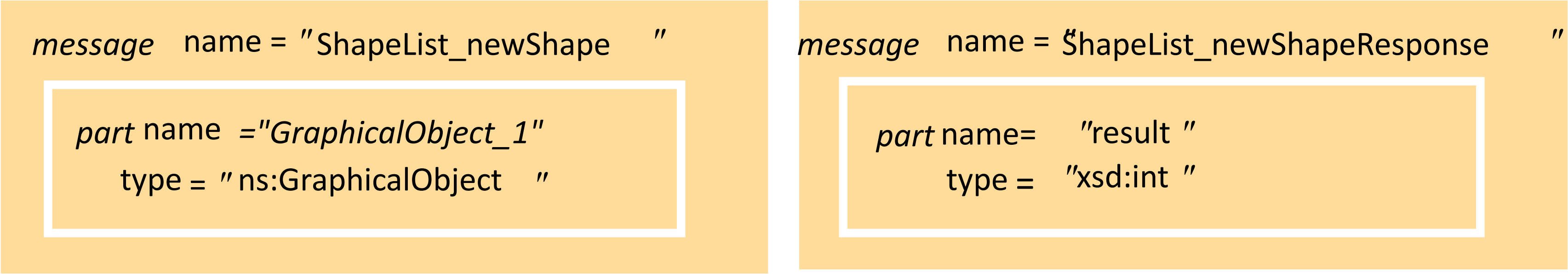
UDDI and WSDL

- UDDI is used to write down the information that became a “row” in the repository (“I have a temperature service…”)
- WSDL documents the interfaces and data types used by the service
- But this isn’t the whole story…

The main elements in a WSDL description



WSDL request and reply messages



message name = ″ShapeList_newShape ″

part name =*"GraphicalObject_1"*
type = ″ns:GraphicalObject ″

message name = ″ShapeList_newShapeResponse ″

part name= ″result ″
type = ″xsd:int ″

tns : target namespace

xsd : XML schema definitions

WSDL operation newShape

operation name = "newShape"
pattern = In-Out

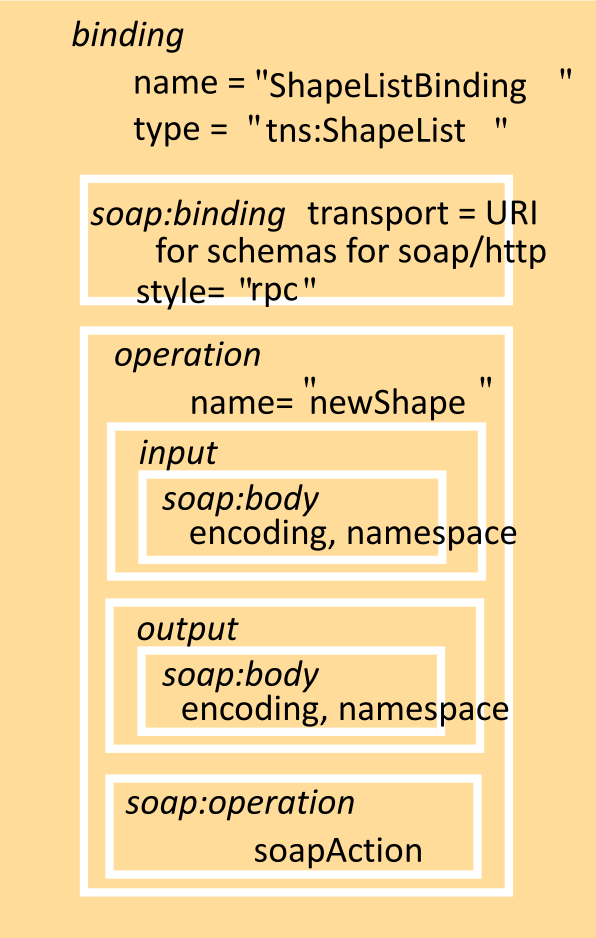
input message = "tns:ShapeList_newShape"

output message = "tns:ShapeList_newShapeResponse"

tns – target namespace xsd – XML schema definitions

The names *operation* , pattern, *input* and *output* are defined in the XML schema for WSDL

SOAP binding and service definitions



binding
name = "ShapeListBinding"
type = "tns:ShapeList"

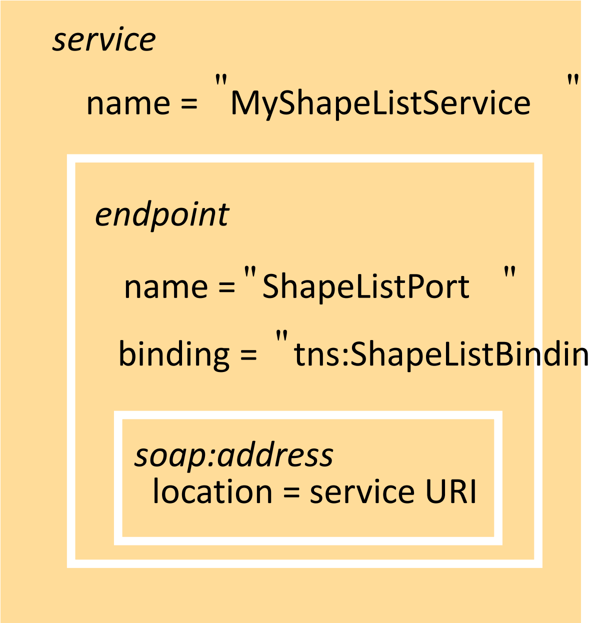
soap:binding transport = URI
for schemas for soap/http
style= "rpc"

operation
name= "newShape"

input
soap:body
encoding, namespace

output
soap:body
encoding, namespace

soap:operation
soapAction



service
name = "MyShapeListService"

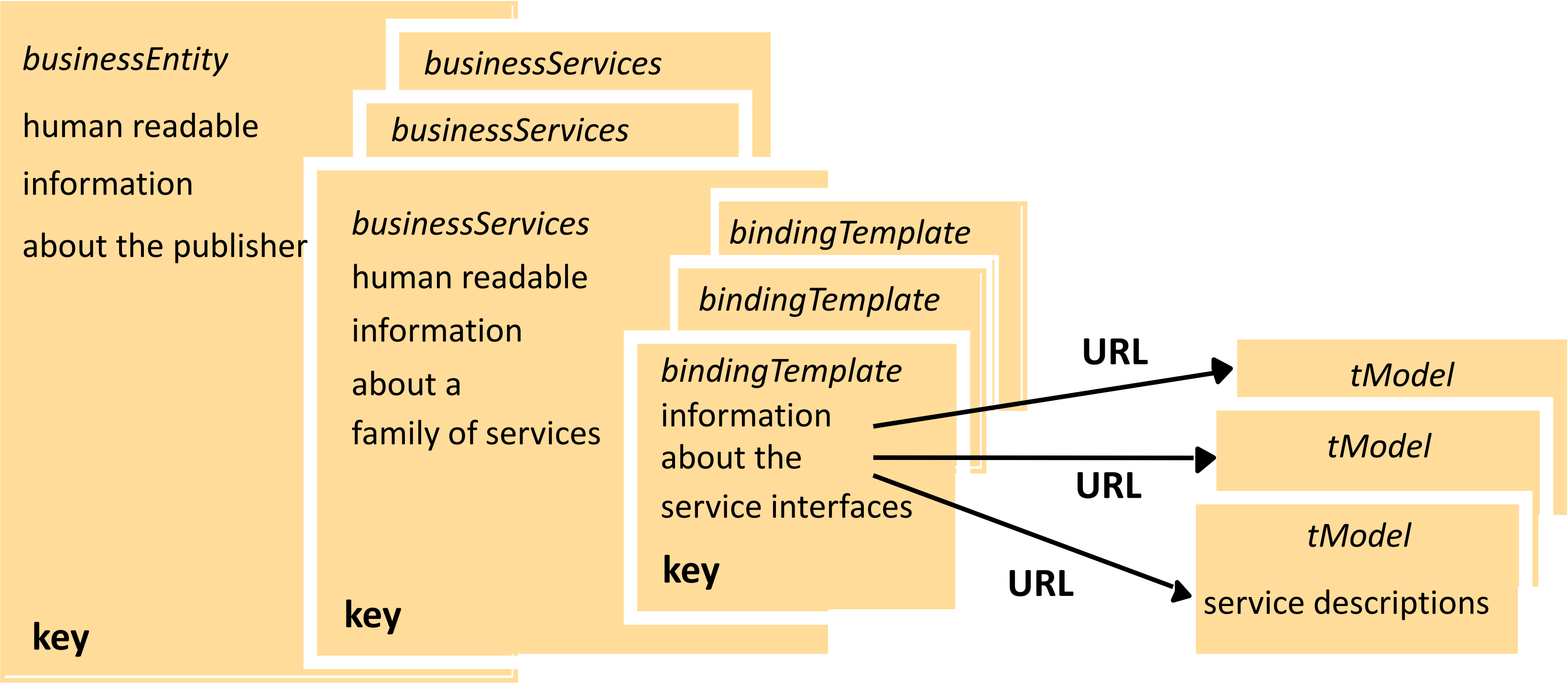
endpoint
name = "ShapeListPort"
binding = "tns:ShapeListBinding"

soap:address
location = service URI

the service URI is:

“http://localhost:8080/ShapeList-jaxrpc/ShapeList”

The main UDDI data structures



So, what’s tricky?

- Web Services doesn’t standardize these four steps, it just assumes that people will hack solutions
- Hence some are hard to implement, we lack standards, and in some cases, solutions are poor ones
- UDDI and WSDL are just a corner of the overall picture!

What about "legacy" applications?

- Some of these Web services are really just front-ends to older legacy applications
  - So to talk to an old IBM database, we might
    - Run the database on some sort of machine, or virtual machine
    - Build one of these translator front-ends
    - And then register *it* with the Web Services router
- This may sound expensive (it is) but it works!
- Obviously, our fancy clustering and load-balancing won't apply to a legacy application, so those fancy tricks are only for "new" code

These are modern challenges

- Web Services can be seen as evolving from prior work
- Most often cited: CORBA, which also was used in many big data centers
- But CORBA didn’t assume that clients came in over the public Internet
  - More often, CORBA was used between a hand-built client and the service it talks to

CORBA approach

- CORBA had what are called
  - Ways to export specialized client stubs
    - The client stub could include server provided decision logic, like “which data center to connect with”
    - Gives data center a form of remote control
  - Factory services: manufacture certain kinds of objects as needed
    - Effect was that “discovery” can also be a “service creation” activity

CORBA is object oriented

- Seems obvious… and it is. CORBA is centered around the notion of an object
  - Objects can be passive (data)
  - … active (programs)
  - … persistent (data that gets saved)
  - … volatile (state only while running)
- In CORBA the application that manages the object is inseparable from the object
  - And the stub on the client side is part of the application
  - The request per-se is an action by the object on itself and could even exploit various special protocols
  - We can’t do this in Web Services

Web Services are document-centric

- Communication is by sending documents (like pages) from client to server and back
- Most guarantees or properties are associated with the document itself, not the service
    - For example, WS_RELIABILITY isn’t about making services reliable, it defines rules for writing reliability requests down and attaching them to documents
    - In contrast, CORBA fault-tolerance standard tells how to make a CORBA service into a highly available clustered service

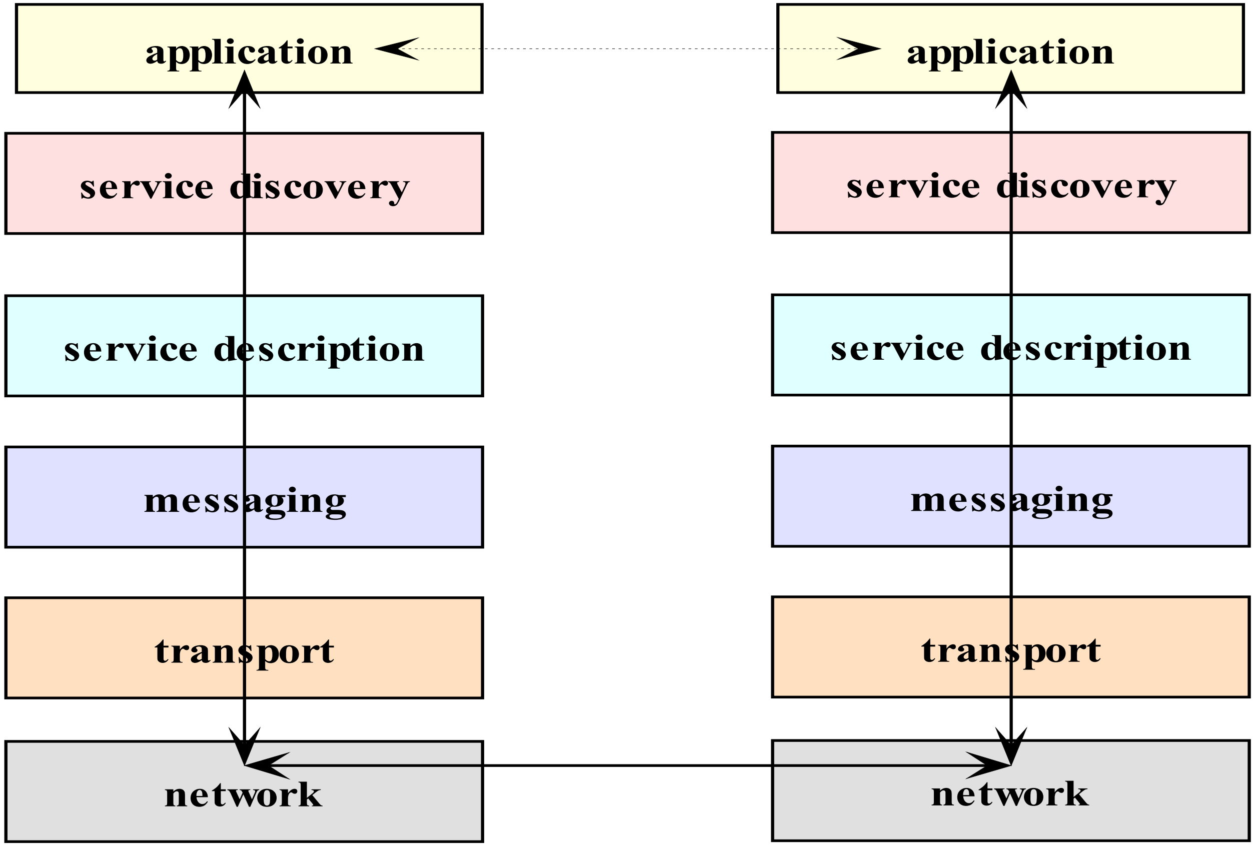
How we do it now

- Client queries directory to find the service
- Server has several options:
  - Web pages with dynamically created URLs
    - Server can point to different places, by changing host names
    - Content hosting companies remap URLs on the fly. E.g. http://www.akamai.com/www.cs.cornell.edu (reroutes requests for www.cs.cornell.edu to Akamai)
  - Server can control mapping from host to IP address
    - Must use short-lived DNS records; overheads are very high!
    - Can also intercept incoming requests and redirect on the fly

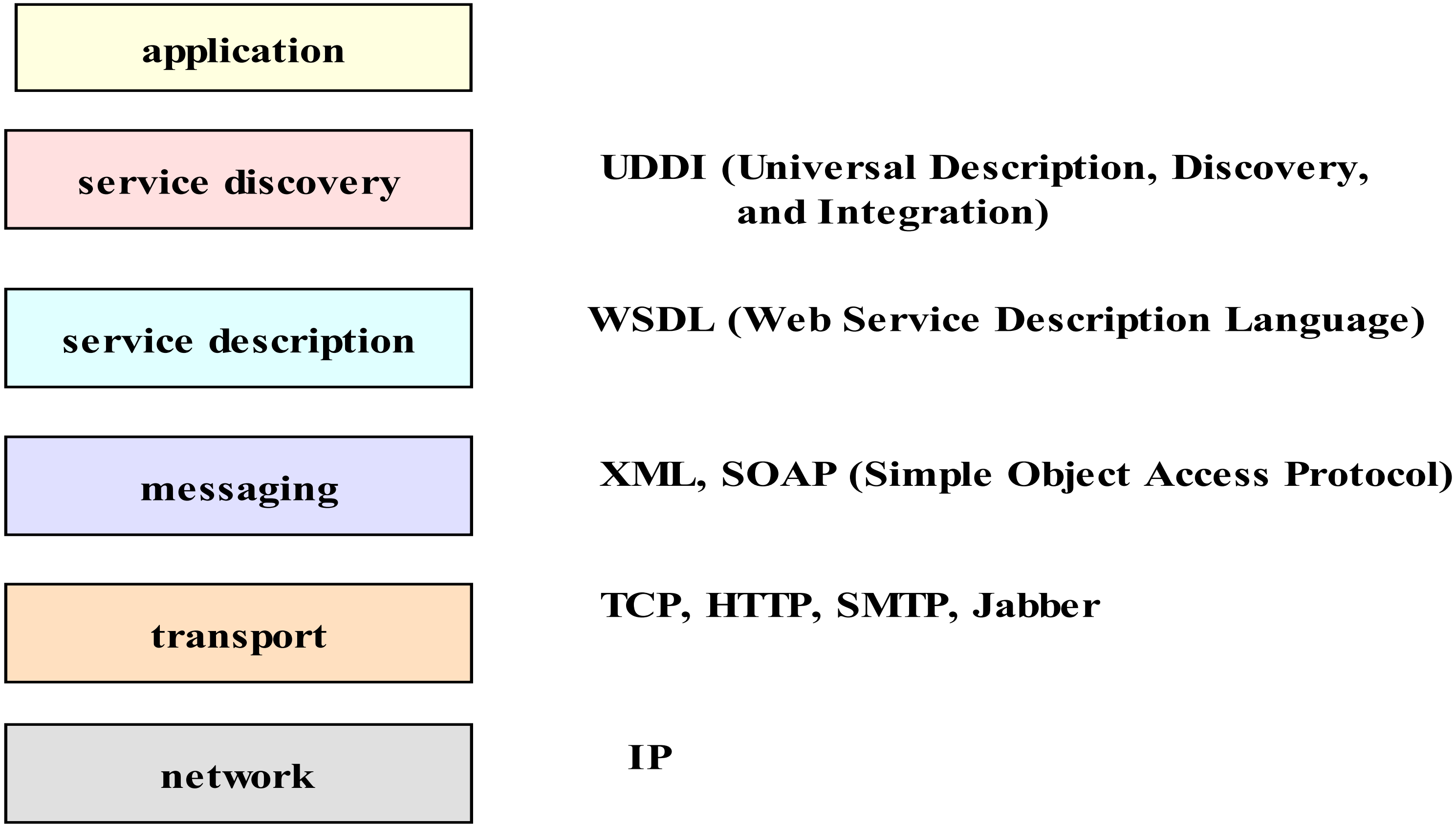
Why this isn’t good enough

- The mechanisms aren’t standard and are hard to implement
  - Akamai, for example, does content hosting using all sorts of proprietary tricks
- And they are costly
  - The DNS control mechanisms force DNS cache misses and hence many requests do RPC to the data center
- We lack a standard, well supported, solution!

Web service protocol stack



Web service protocols

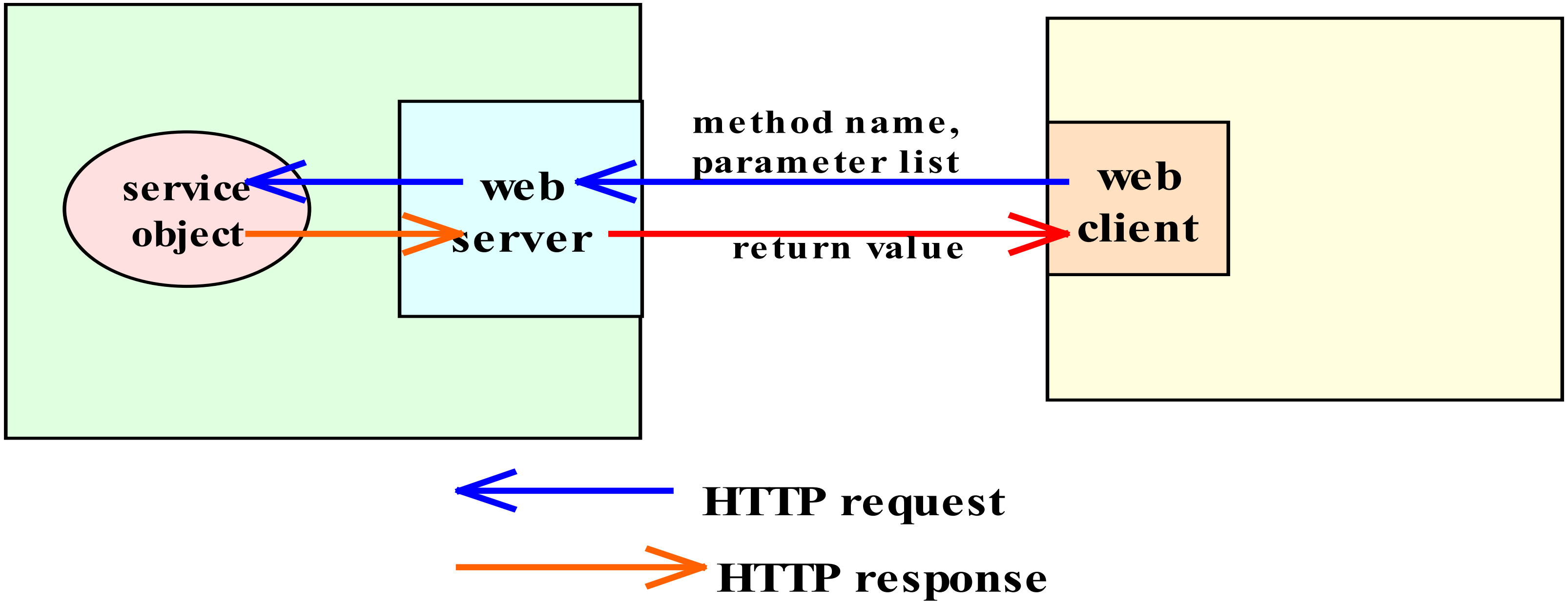


application	
service discovery	UDDI (Universal Description, Discovery, and Integration)
service description	WSDL (Web Service Description Language)
messaging	XML, SOAP (Simple Object Access Protocol)
transport	TCP, HTTP, SMTP, Jabber
network	IP

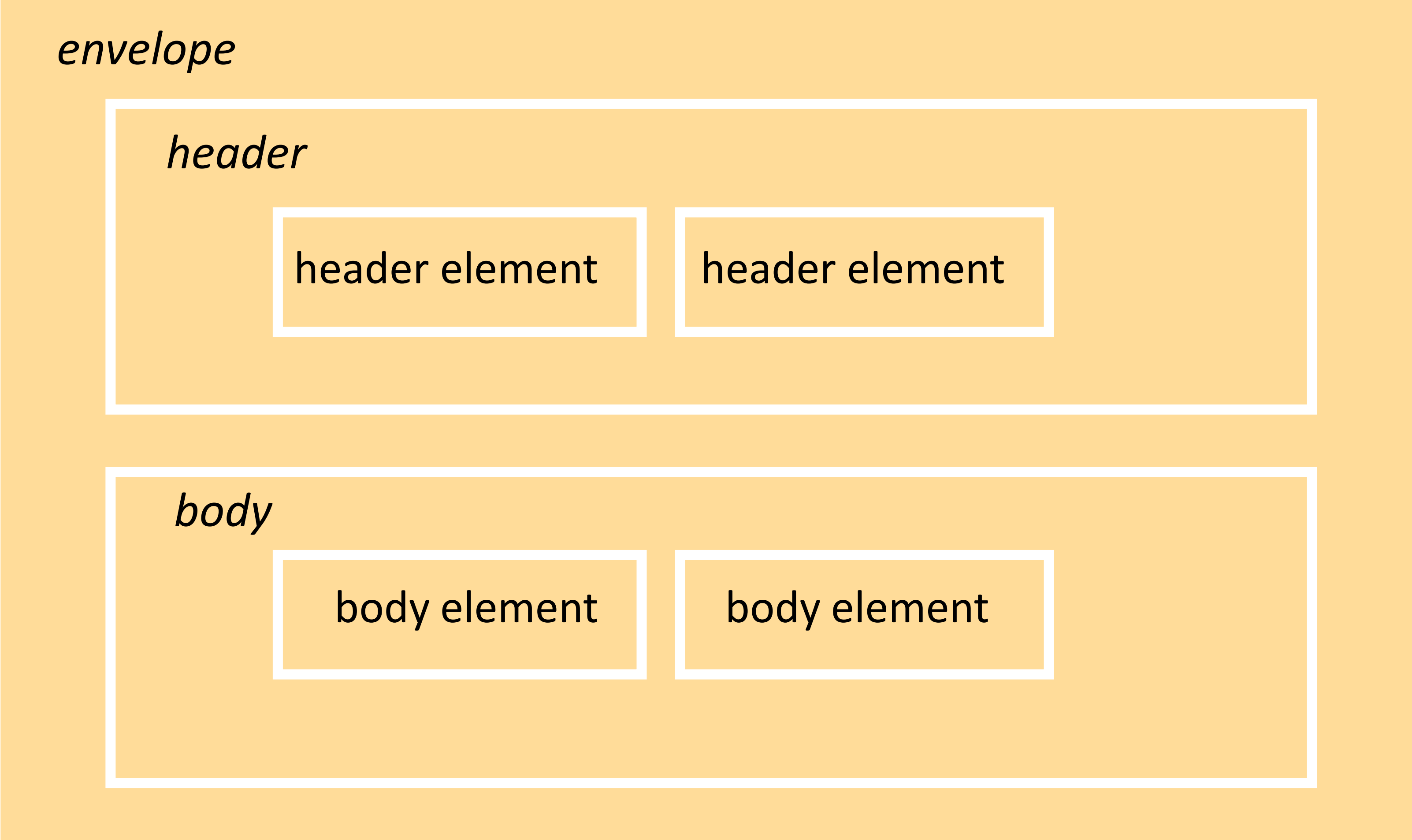
SOAP

- SOAP is a protocol which applies XML for message exchange in support of remote method calls over the Internet.
- Compared to remote method invocation or CORBA-based facilities:
  - SOAP is web-based or “wired” and hence is not subject to firewall restrictions
  - Language-independent
  - Can provide just-in-time service integration

Remote Procedure Call using HTTP



SOAP message in an envelope



envelope

header

header element

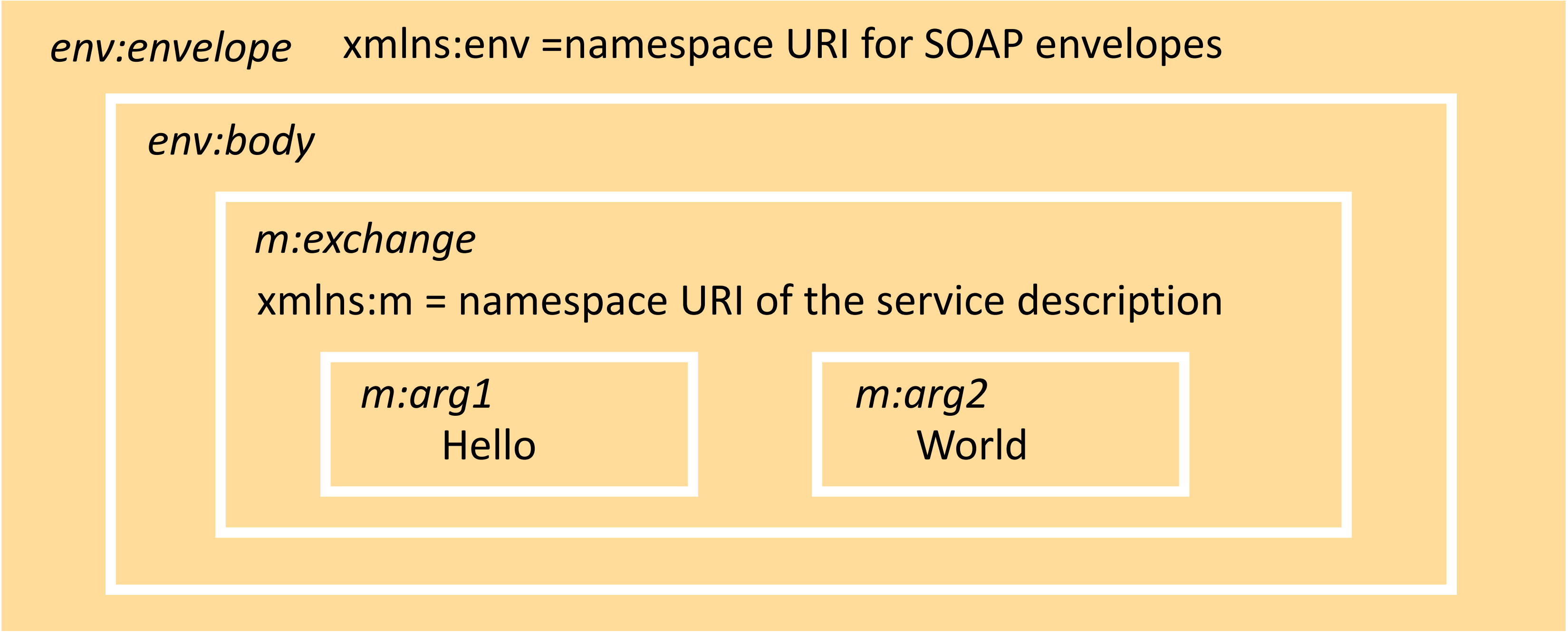
header element

body

body element

body element

Example of a simple request without headers



env:envelope xmlns:env =namespace URI for SOAP envelopes

env:body

m:exchange

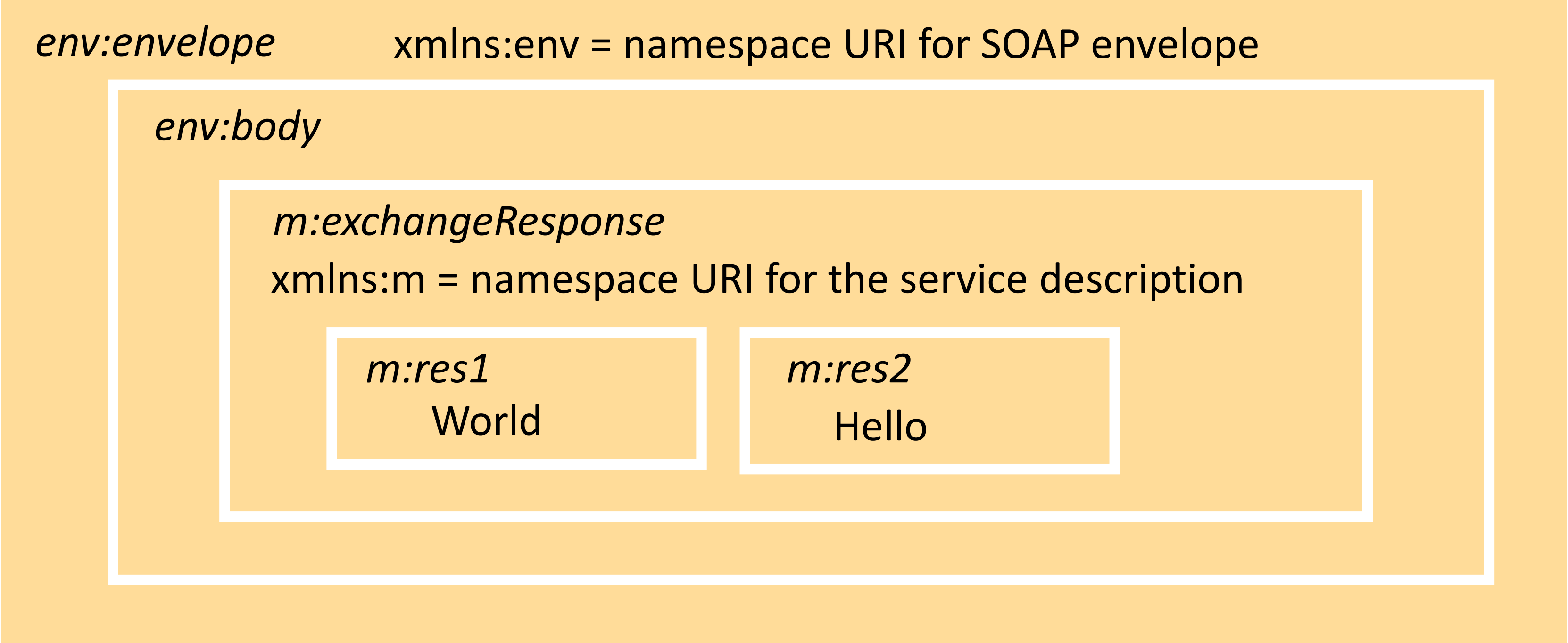
xmlns:m = namespace URI of the service description

m:arg1
Hello

m:arg2
World

In this figure and the next, each XML element is represented by a shaded box with its name in italic followed by any attributes and its content

Example of a reply



env:envelope xmlns:env = namespace URI for SOAP envelope

env:body

m:exchangeResponse

xmlns:m = namespace URI for the service description

m:res1
World

m:res2
Hello

HTTP POST Request in SOAP client-server communication

```
POST /examples/stringer
Host: www.cdk4.net
Content-Type: application/soap+xml
Action: http://www.cdk4.net/examples/stringer#exchange


<env:envelope xmlns:env=  namespace URI for SOAP envelope>
<env:header> </env:header>
<env:body> </env:body>
</env:Envelope>
```

endpoint address

action

HTTP header

Soap message

An Example SOAP Request

source: (http://www.soaprpc.com/tutorials/) A Busy Developer’s Guide To Soap1.1

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/1999/XMLSchema"
 xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
    <SOAP-ENV:Body>
          <m:getStateName xmlns:m="http://www.soapware.org/">
               <statenum xsi:type="xsd:int">41</statenum>
          </m:getStateName>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

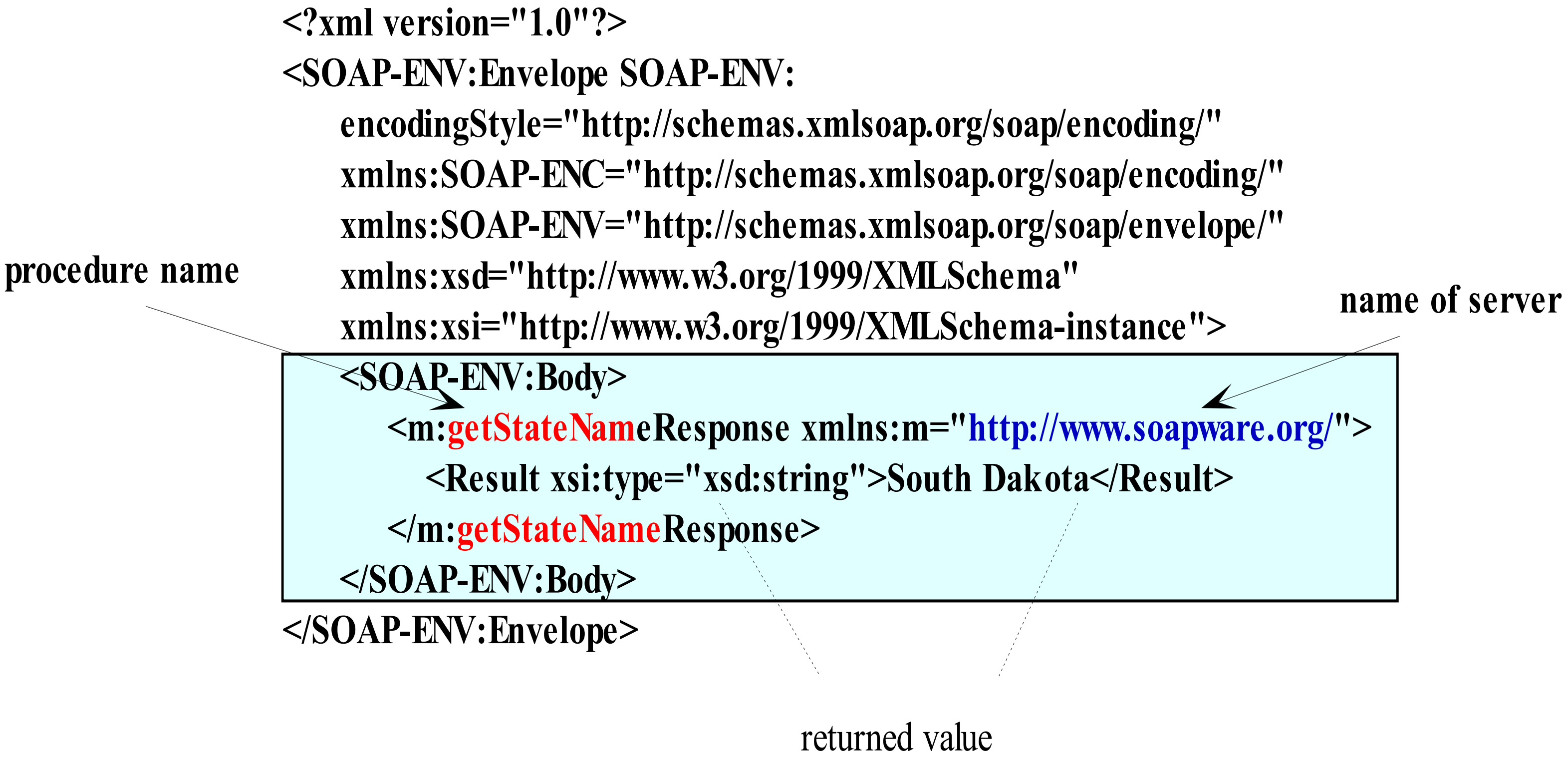
procedure name

name of server

parameter of type int and value 41

Response example

source: (http://www.soaprpc.com/tutorials/) A Busy Developer’s Guide To Soap1.1



```
<?xml version="1.0"?>
<SOAP-ENV:Envelope SOAP-ENV:
    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
    <SOAP-ENV:Body>
       <m:getStateNameResponse xmlns:m="http://www.soapware.org/">
          <Result xsi:type="xsd:string">South Dakota</Result>
       </m:getStateNameResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

procedure name

name of server

returned value

HTTP and SOAP RPC Request

source: (http://www.soaprpc.com/tutorials/) A Busy Developer’s Guide To Soap1.1

A SOAP message can be used to transport a SOAP remote procedure request/response, as follows:

```
POST    /examples    HTTP/1.1
User-Agent: Radio UserLand/7.0 (WinNT)
Host: localhost:81
Content-Type: text/xml; charset=utf-8
Content-length: 474
SOAPAction: "/examples"
<blank line>
<text for SOAP message>
```

An HTTP request that carries a SOAP RPC request

source: (http://www.soaprpc.com/tutorials/) A Busy Developer’s Guide To Soap1.1

```
POST    /examples    HTTP/1.1
User-Agent: Radio UserLand/7.0 (WinNT)
Host: localhost:81
Content-Type: text/xml; charset=utf-8
Content-length: 474
SOAPAction: "/examples"

<?xml version="1.0"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/" xmlns:
SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:
SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
<SOAP-ENV:Body>
       <m:getStateName xmlns:m="http://www.soapware.org/">
         <statenum xsi:type="xsd:int">41</statenum>
         </m:getStateName>
 </SOAP-ENV:Body>
 </SOAP-ENV:Envelope>
```

An HTTP request that carries a SOAP RPC response

source: (http://www.soaprpc.com/tutorials/) A Busy Developer’s Guide To Soap1.1

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 499
Content-Type: text/xml; charset=utf-8
Date: Wed, 28 Mar 2001 05:05:04 GMT
Server: UserLand Frontier/7.0-WinNT

<?xml version="1.0"?>
<SOAP-ENV:Envelope SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
  <SOAP-ENV:Body>
    <m:getStateNameResponse xmlns:m="http://www.soapware.org/">
      <Result xsi:type="xsd:string">South Dakota</Result>
      </m:getStateNameResponse>
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

SOAP data types

- SOAP Data Types

  http://download.oracle.com/docs/cd/E14154_01/dev.835/e12868/e01_soap_types.htm
- Uses XML Schema data types
- Primitive Types

  string, boolean, decimal, float, double, duration, dateTime, time, date, gYearMonth, gYear, gMonthDay, gDay, gMonth, hexBinary, base64Binary, anyURI, QName, NOTATION

SOAP data types

Derived Types

- Simple types (derived from a single primitive type)
  * integer is derived from decimal
  * int (-2147483648 <= int <= 2147483647) is derived from long which is derived from integer
  * 5-digit zip code can be derived from int
  * may use regular expressions to specify derived types, such as ([A-Z]){2,3}-\d{5}

Complex Type

Complex types (struct or array)

- Struct example

```
<instructor>
<firstname xsi:type="xsd:string">Ed</firstname>
<lastname xsi:type="xsd:string">Donley</lastname>
</instructor>
```

- Array example

```
<mathcourses xsi:type=
    "SOAP-ENC:Array" SOAP ENC:arrayType="se:string[3]">
    <se:string>10452C</se:string> <se:string>10454C</se:string>
    <se:string>11123T</se:string>
</mathcourses>
```

SOAP Service Parameter Types

- All Java primitive types and their corresponding wrapper classes
- Java arrays
- java.lang.String
- java.util.Date
- java.util.GregorianCalendar
- java.util.Vector
- java.util.Hashtable
- java.util.Map
- java.math.BigDecimal
- javax.mail.internet.MimeBodyPart
- java.io.InputStream
- javax.activation.DataSource
- javax.activation.DataHandler
- org.apache.soap.util.xml.QName
- org.apache.soap.rpc.Parameter
- java.lang.Object (must be a JavaBean)

Building Web Services with JAX-WS

- JAX-WS stands for Java API for XML Web Services.
  - JAX-WS is a technology for building web services and clients that communicate using XML.
  - JAX-WS allows developers to write message-oriented as well as RPC-oriented web services.
- In JAX-WS, a remote procedure call is represented by an XML-based protocol such as SOAP.
  - The SOAP specification defines the envelope structure, encoding rules, and conventions for representing remote procedure calls and responses.
  - These calls and responses are transmitted as SOAP messages (XML files) over HTTP.

Features of JAX-WS

- With JAX-WS, clients and web services have a big advantage: the platform independence of the Java programming language.
- JAX-WS is not restrictive: a JAX-WS client can access a web service that is not running on the Java platform, and vice versa.
  - This flexibility is possible because JAX-WS uses technologies defined by the World Wide Web Consortium (W3C): HTTP, SOAP, the Web Service Description Language (WSDL).
- JAX-WS 2.0 supports the Web Services Interoperability.

Basic steps for creating the web service and client

1. Code the implementation class.
2. Compile the implementation class.
3. Use **`wsgen`** to generate the artifacts required to deploy the service.
4. Package the files into a WAR file.
5. Deploy the WAR file. The tie classes (which are used to communicate with clients) are generated by the Application Server during deployment.
6. Code the client class.
7. Use **`wsimport`** to generate and compile the stub files.
8. Compile the client class.
9. Run the client.

Hello World Service Endpoint Implementation Class

```
package helloservice.endpoint;

import javax.jws.WebService;

@WebService()
public class Hello {
    private String message = new String("Hello, ");

    public void Hello() {}

    @WebMethod()
    public String sayHello(String name) {
            return message + name + ".";
    }
}
```

Hello Client

```
package simpleclient;

import javax.xml.ws.WebServiceRef;
import helloservice.endpoint.HelloService;
import helloservice.endpoint.Hello;

public class HelloClient {
    @WebServiceRef(wsdlLocation="http://localhost:8080/helloservice/hello?wsdl")
    static HelloService service;

    public static void main(String[] args) {
        try {
          HelloClient client = new HelloClient();
          client.doTest(args);
        } catch(Exception e) {
        e.printStackTrace();
    }
}
```

Hello Client, Continued

```
public void doTest(String[] args) {
        try {
                System.out.println("Retrieving the port from the following service: " +
                        service);
                Hello port = service.getHelloPort();
                System.out.println("Invoking the sayHello operation on the port.");

                String name;
                if (args.length > 0) {
                                name = args[0];
                } else {
                                name = "No Name";
                }

                String response = port.sayHello(name);
                System.out.println(response);
        } catch(Exception e) {
        e.printStackTrace();}}}
```